Naval Research Laboratory

Stennis Space Center, MS 39529-5004

NRL/FR/7430--04-10,039

Summary Report on a Seamap-C Chirp Deconvolution Algorithm with Demonstrations Using Synthetic and Field Data

DENNIS A. LINDWALL

Seafloor Sciences Branch
Marine Geosciences Division

September 13, 2004

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
13-09-2004	NRL Formal Report	01-04-1999 to 30-01-2000

4. TITLE AND SUBTITLE

Summary Report on a Seamap-C Chirp Deconvolution Algorithm with Demonstrations Using Synthetic and Field Data

5a. CONTRACT NUMBER
N0001402WX30017

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62435N

6. AUTHOR(S)

Dennis A. Lindwall

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
74-6632-00

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Marine Geosciences Division
Stennis Space Center, MS 39529-5004

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/FR/7430--04-10,039

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217-5660

10. SPONSOR / MONITOR'S ACRONYM(S)
ONR

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes several ways to generate a chirp signal for the Seamap-C side-scan sonar system and then to remove the chirp signal from the field data to produce high-resolution seafloor images. Synthetic data are generated using one of several chirps according to the system specifications, data collection methods, and preliminary processing of the Seamap-C system. These synthetic data files, as well as field data, are read by the deconvolution program, which removes the known source signal. Several chirp signals are compared to demonstrate the tradeoffs between resolution and noise levels.

15. SUBJECT TERMS

Seamap-C
Side-scan sonar system
Chirp signal
Synthetic data

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dennis A. Lindwall
Unclassified	Unclassified	Unclassified	UL	47	19b. TELEPHONE NUMBER *(include area code)* (228) 688-5306

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

CONTENTS

SUMMARY REPORT ON A SEAMAP-C CHIRP DECONVOLUTION ALGORITHM WITH DEMONSTRATIONS USING SYNTHETIC AND FIELD DATA

BACKGROUND

Side-scan sonar systems were first developed years ago to image seafloor geological features. The University of Hawaii developed the ability to also measure bathymetry with their SeaMarc side-scan system by using two rows of transducers on each side and measuring the angle of the reflected return. Since these early systems, there have been many side-scan systems using the same principles but with different sizes and frequencies for different applications. All of these systems transmit a narrowband sound pulse (ping) as the source. The spatial resolution of a side-scan sonar system using a narrowband ping is determined by the length of the ping and by the beam pattern from the transducer array. A very short ping will give the highest resolution. For example, the Seamap-C system typically uses pings from 2- to 10-ms long. A 2-ms ping gives a resolution along the swath of 1.5 m for a seafloor grazing ray and 1.06 m at a 45 deg slant angle. A 10-ms pulse has a resolution of about 5 m at a 45 deg slant range. These resolutions are theoretical, ignoring the effects of noise and signal strength.

Using a broadband chirp signal has two advantages regarding resolution and power over a continuous wave (CW) ping signal. The theoretical resolution limit of a broadband signal, according to the Rayleigh criterion, is approximately

$$\frac{2}{3\nu_b} = \frac{2}{3\left(V/R\right)} = \frac{2R}{3V},$$

where ν_b is the frequency bandwidth, V is the sound speed, and R is the resolution. This leads to a resolution of

$$R = V/\nu_b$$

which is, with a typical sound speed of 1,500 m/s and a bandwidth of 2 kHz, 0.75 m for a seafloor grazing ray or about 0.5 m at a 45 deg slant angle. This resolution does not depend on the length of the signal or its frequency, only the bandwidth, so a short chirp and a long chirp of the same bandwidth have the same theoretical resolution. The Seamap-C system can generate a chirp of up to 1 second long or 500 times as long as the shortest ping. A 1-second, 2-kHz bandwidth chirp gives not only twice the resolution but 500 times the power as the 2-ms ping. Using the Seamap-C system with a chirp signal combines the high resolution of smaller, high-frequency systems and the long range capability of larger, low-frequency systems.

Datasonics, Inc. makes a commercially available chirp side-scan sonar system (Parent et al. 1993). This system uses a hardware deconvolution before the data are recorded. They also use a Gaussian frequency window to reduce the deconvolution side lobes.

Manuscript approved February 20, 2004.

SYNTHETIC DATA

Synthetic data are useful for testing the processing algorithms since it will be free of any noise, artifacts, or errors, and the environment can be simple and precisely known. Comparing processed synthetic data with the known environment should give exact agreement. The synthetic data program can be expanded to include off-beam reflections and the response of both arrays from a known bathymetry. The synthetic data are written in the format that the Seamap-C system software writes and records data. This way every program that reads and processes the field data can be tested with synthetic data.

The synthetic data algorithm produces the time series output from a single array along its beam axis with a chirped signal, either linear sweep or an arccosine sweep. The environmental response appears as reflections at different times and strengths. A linear sweep has a frequency that changes at a constant rate throughout the period of the sweep and is calculated thus:

$$a(t) = \sin\left(2\pi(\omega_1 + t(\omega_2 + \omega_1)/(2t_c))\right) \tag{1}$$

where ω_1 is the start frequency, ω_2 is the end frequency, and t_c is the length of the chirp signal in seconds. This signal needs to be windowed to reduce the deconvolution side lobes. The program includes several time-domain windows as options. The synthetic data algorithm also includes some nonlinear chirps that do not need time-domain windowing so that the transducers can be driven at full power for the entire signal duration. One nonlinear chirp signal is produced by:

$$a = \sin\left(2\pi(\omega_1 + (\omega_2 - \omega_1)a\cos(1 - t/t_c)/\pi)\right). \tag{2a}$$

This one has a slightly asymmetric spectrum with one sharp edge, yet its autocovariance function has small side lobes. Another nonlinear chirp signal is:

$$a(t) = \sin\left(2\pi(\omega_1 + (\omega_2 - \omega_1)a\cos(1 - t/2t_c)/2\pi)\right). \tag{2b}$$

Data from the Seamap-C system are compressed before being recorded by shifting the center frequency of the signal (basebanding) and calculating the analytical waveform at a much lower sampling rate (quadrature decimation). The basebanding is done using:

$$a(t) + ib(t) = \left[\sqrt{2}x(t)\cos(\omega_0 t)\right] * h(t) + i\left[\sqrt{2}x(t)\sin(\omega_0 t)\right] * h(t) \tag{3}$$

where * denotes convolution, $x(t)$ is the original time series data, ω_0 is the center frequency of the chirp signal, $h(t)$ is an ideal low-pass filter, and $a(t) + ib(t)$ is the resulting analytic (complex) signal. The resulting signal is low-pass filtered and decimated to a sampling rate of 2,875 Hz. This is less than twice the Nyquist frequency, but is adequate since an analytic signal is recorded. In the engineering terminology, the real part of the analytic signal is *I* (in phase) and the imaginary part is *Q* (quadrature) in the engineering terminology of *I* and *Q* pairs.

The synthetic data are produced by adding several chirp signals (a 10.5- to 12.5-kHz linear sweep) at different amplitudes and time delays. This is equivalent to convolving a simple Green's function (Earth response to an impulse) with the chirp signal. The real-time signal at the original 46-kHz sampling rate is saved as a file. This signal is changed to an analytic signal by using the Hilbert transform of the real signal for the imaginary part. This analytic signal is frequency shifted (basebanded) using Eq. (3), low-pass filtered, and decimated by a factor of 16. The real and imaginary parts of the resulting signal, *a* and *b* of Eq. (3), are

saved as two separate files. These files simulate the data recorded in the field from one transducer array of the Seamap-C system.

DECONVOLUTION

The long chirp signal must be removed from the data by deconvolution or some other source signal removal operation. Deconvolution of two signals is equivalent to convolving one signal with the inverse of the other. For a source signal $v(t)$ and Earth response $e(t)$ the recorded signal is

$$g(t) = v(t) * e(t). \tag{4}$$

We can achieve the desired effect of deconvolution by convolving the recorded signal with the source, thus avoiding any division by zero problems:

$$\Phi_{vg}(t) = g(t) * \bar{v}(-t) = (v(t) * e(t)) * \bar{v}(-t) \tag{5}$$

$$\begin{aligned} &= e(t) * (v(t) * \bar{v}(-t)) \\ &= e(t) * \Phi_{vv}(t), \end{aligned} \tag{6}$$

where Φ_{vv} is the autocorrelation function, and $\bar{v}$ denotes the complex conjugate of v. Φ_{vv}, for a broadband signal, has the same time resolution as deconvolution. Convolution in the time domain is

$$g_1(t) * g_2(t) = \int_{-\infty}^{+\infty} g_1(\tau) g_2(t-\tau) d\tau, \tag{7}$$

which is similar to the cross covariance

$$g_1(t) \otimes g_2(t) = \int_{-\infty}^{+\infty} g_1(\tau) \bar{g}_2(t+\tau) d\tau, \tag{8}$$

where $\otimes$ denotes cross covariance. So, convolution is equivalent to cross covariance with one of the time series reversed. This assumes that we know the exact signal that is transmitted. This will not be true in the field. While we will know the voltage and current in the transducer array during the transmit event, the actual acoustic signal in the water will be different because of nonlinear effects within the transducers. We may be able to measure these effects from calibration tests, or we may be able to back it out of field data by an inversion process (e.g. Wood 1999).

The recorded data are in a different form than the transmitted or received acoustic signals. The base-banding and quadrature decimation are described in the Synthetic Data section. The data and source can be reconstructed at the original sampling rate and frequency and then used to calculate the cross covariance, but this is unnecessary and slow but useful as a sanity check. Two better alternatives are to partially reconstruct the data and the transmitted signal as real-time series with a low center frequency, and then calculate the cross covariance, or else to deconvolve the complex, basebanded, quadrature-decimated data directly.

The real data can be partially reconstructed by resampling the complex, basebanded signal at twice the rate then shifting the center frequency by

$$b(t) = \sqrt{2} I(t) \cos\left(2\pi t \Delta/4\right) + \sqrt{2} Q(t) \sin\left(2\pi t \Delta/4\right), \tag{9}$$

where Δ is the new sampling rate, so shifting the bandwidth by $\Delta/4$ (half the Nyquist frequency) places the lower edge of the frequency band at zero. The recorded source signal can be partially reconstructed in the same way and then convolved with $b(t)$ to recover the Green's function. This recovered Green's function should be identical, except for the sampling rate, as that recovered from the original data and chirped source

signal (a in Eqs. (1) and (2)). The real-time series $b(t)$ has the same number of samples as the original $I(t)$ and $Q(t)$ pairs, so the convolution calculation should be as fast as the complex convolution.

The recorded data and source signal in the complex, basebanded, quadrature-decimated form can be deconvolved directly without any frequency shifting by convolving the complex data and signal (I and Q pairs). Complex convolution involves cross terms; some analysis programs may do this properly, but many analysis packages do not, and the algorithm must be correctly written. If Φ_{ds} is the cross covariance of the complex data d and source s, then the real and imaginary parts of Φ_{ds} are

$$\Phi\{real\}_{ds} = \Phi_{d\{real\}s\{real\}} + \Phi_{d\{imaginary\}s\{imaginary\}} \tag{10}$$

$$\Phi\{imaginary\}_{ds} = \Phi_{d\{real\}s\{imaginary\}} - \Phi_{d\{imaginary\}s\{real\}} \,. \tag{11}$$

As shown in the bottom plot of Fig. 4, the complex signal Φ_{ds} has a central frequency of 0 Hz, so a real-time representation of this signal will only have a bandwidth of 1 kHz. To display the data with its full bandwidth and resolution, the sampling rate must be doubled, and the center frequency must be shifted as in Eq. (9). The Seamap-C system has a decimated, basebanded sampling rate of 2,875 Hz, so the frequency shift in Eq. (9) ($\Delta/4$) can be half that amount in order to center the spectrum in the new bandwidth.

DEMONSTRATIONS

A 5-ms linear chirp from 10.5 to 12.5 kHz is shown in Fig. 1, with its basebanded form in Fig. 2 demonstrating the difference in the frequencies and minimum sampling rate. The time series in Fig. 1 has a 46-kHz sampling rate (the Seamap-C system sampling rate), which is nearly twice the Nyquist rate, but still reasonable for this signal. The basebanded signal is easily sampled at 2,875 Hz, which is what the Seamap-C system records. Figure 3 shows the 50-ms long chirp used for both the synthetics and the field test with the real-time series, sweeping from 10.5 to 12.5 kHz, at the top and the basebanded form on the bottom. The spectra of both the time series and the basebanded signal (Fig. 4) have the same shape but different central frequencies. The linear chirp in Fig. 3 is used to generate synthetic data with five simple reflections as a Green's function (Fig. 5). Figure 6 shows the real (I or "in phase") and imaginary (Q or "quadrature") parts of the complex, synthetic data signal. These are called the I and Q pairs and are the data that are recorded in the field by the Seamap-C system. Lastly, the data are deconvolved by calculating the cross covariance of the simulated data with the chirp signal in both the high-frequency real format and the basebanded format. The two recovered Green's functions from the original 46-kHz signal and the decimated, basebanded signal are compared in Figs. 7 and 8. Note that there is no window applied to the chirp signal, which causes the rough spectra (Fig. 4) and the large side lobes in the deconvolved data (Figs. 7 and 8). These noise effects are caused by the sharp edges in the time-domain signal (Figs. 1, 2, and 3) and can be reduced by either applying a time-domain window to the linear chirp or by using a nonlinear chirp.

The time-domain window used for this example is a quarter-cosine window. There are other windows that give smaller side lobes for the autocorrelation function, but the quarter-cosine window has an autocorrelation central peak that is only slightly wider than that for the unwindowed signal. Shown are the windowed 46-kHz signal (top of Fig. 9), the basebanded, windowed signal (bottom of Fig. 9), and the spectra of the two (Fig. 10). Note how much smoother the spectra of the windowed signals are (Fig. 10) than the unwindowed signals (Fig. 4). New synthetic data were generated using the same five reflections as before in Fig. 4, but using the windowed signals (Fig. 9), and are shown as both the 46-kHz data and the basebanded data (Fig. 11). The data were then deconvolved by calculating the cross covariance of the new simulated data and the windowed chirp signals in both the 46-kHz real format and the basebanded format. The data are shown as the two recovered Green's functions from the 46-kHz signal (top of Fig. 12) and the decimated, basebanded signal (bottom of Fig. 12). The side lobes are much smaller than for the unwindowed case, and the smallest reflection is clearly distinguished from the side lobes.

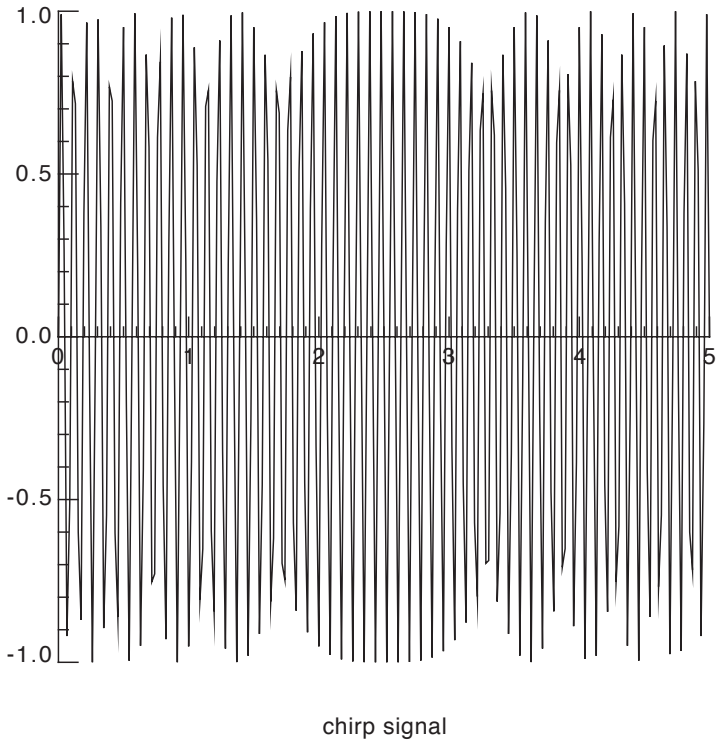

Fig. 1 — Time series of a linear 5-ms long chirp signal sweeping from 10.5 to 12.5 kHz. The horizontal axis is milliseconds and the sampling rate is 46 kHz. The period clearly decreases from left to right. There are just over 10.5 cycles in the first ms and just under 12.5 cycles in the last ms. The apparent amplitude variations are a digitation effect. The chirp signal is normally much longer than 5 ms with the Seamap-C system (up to 1 full second) but the individual cycles would not be resolvable on a page-sized plot.

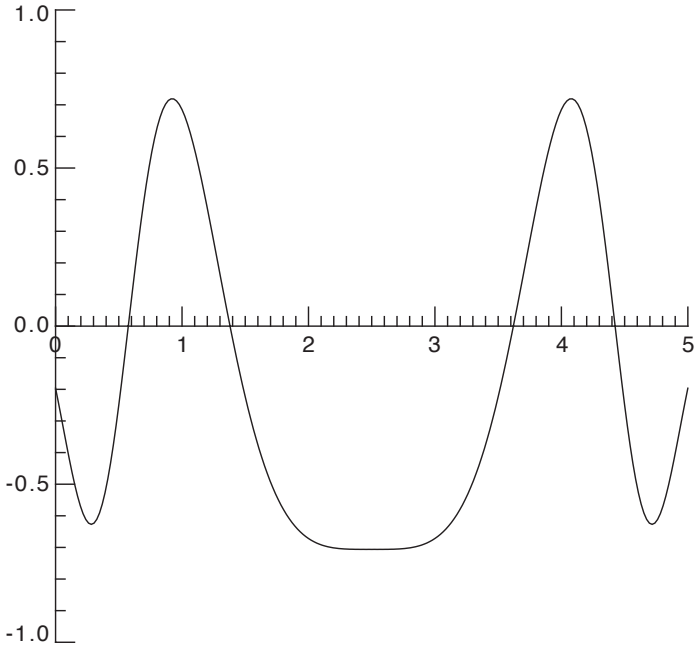

Fig. 2 — Basebanded signal from Fig. 1. This is the real part of the modulated signal removed from the 11.5-kHz carrier signal as calculated by Eq. (3). This chirp signal sweeps from -1 to +1 kHz. The $\sqrt{2}$ amplitude reduction is a normalization factor in Eq. (3).

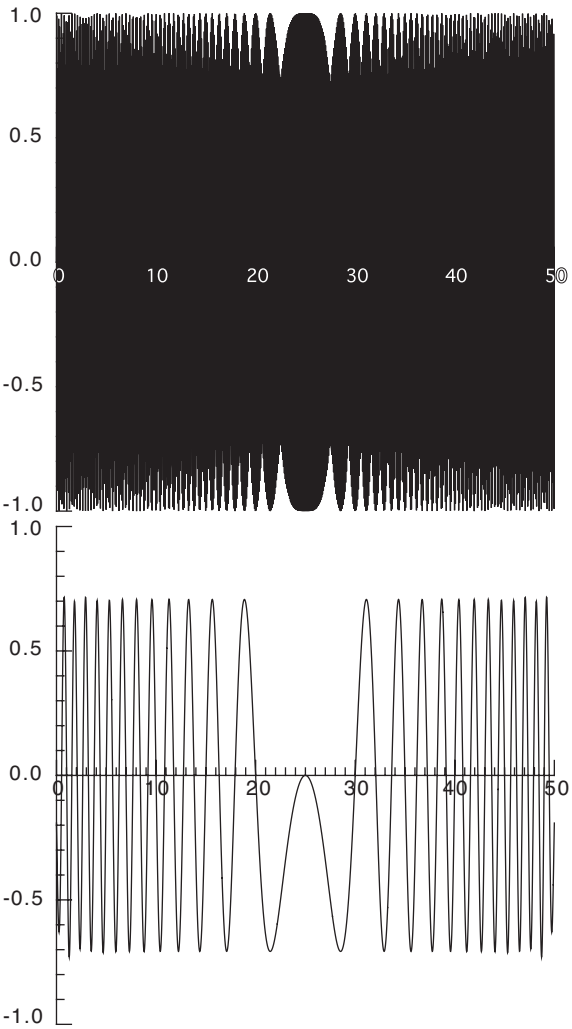

Fig. 3 — The original chirp source time series and the real part (*I*) of the basebanded source. This chirp sweeps from 10.5 to 12.5 kHz as in Figs. 1 and 2, but this is 50-ms long and will be used in the next examples.

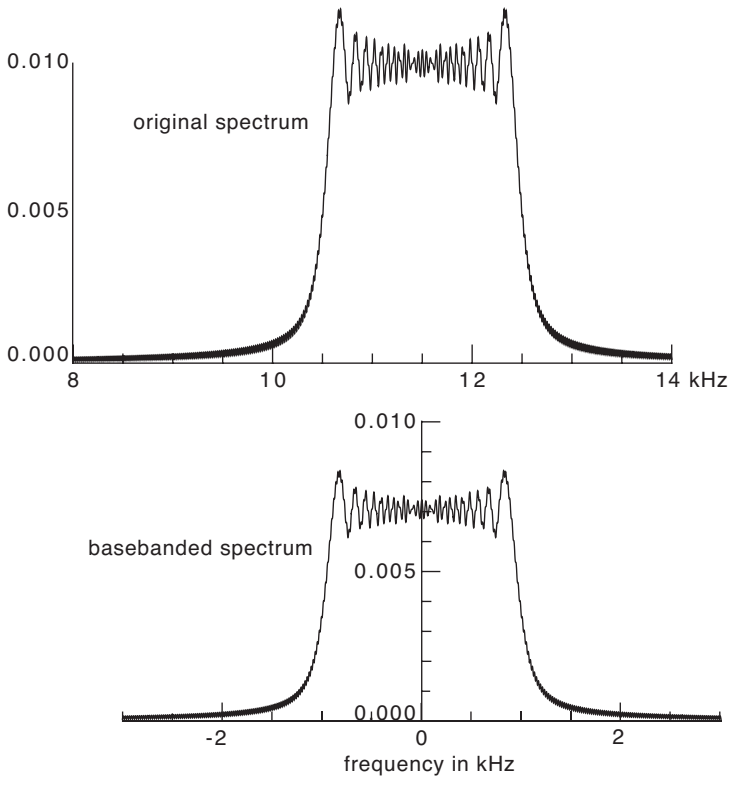

Fig. 4 — Spectra of the original 10.5 to 12.5 kHz chirp (top) and the basebanded chirp with a frequency range from –1.0 to +1.0 kHz (bottom). The lower amplitude of the basebanded spectrum is due to the $\sqrt{2}$ factors in Eq. (3).

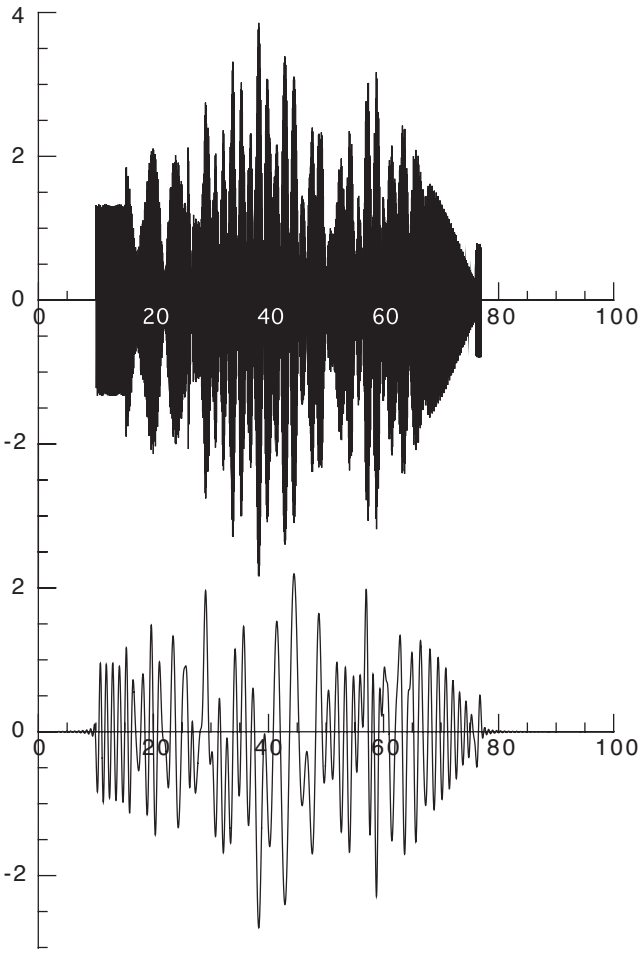

Fig. 5 — The original time series data (top) and the basebanded data (bottom). These data are calculated by adding a source time series for each of several impulses. The source chirp is 50-ms long as in Fig. 3. The impulses are at 10.0, 15.2, 18.0, 26.0, and 27.0 ms. Each has a different amplitude and are arranged so that the chirp signals overlap and interfere by varying degrees.

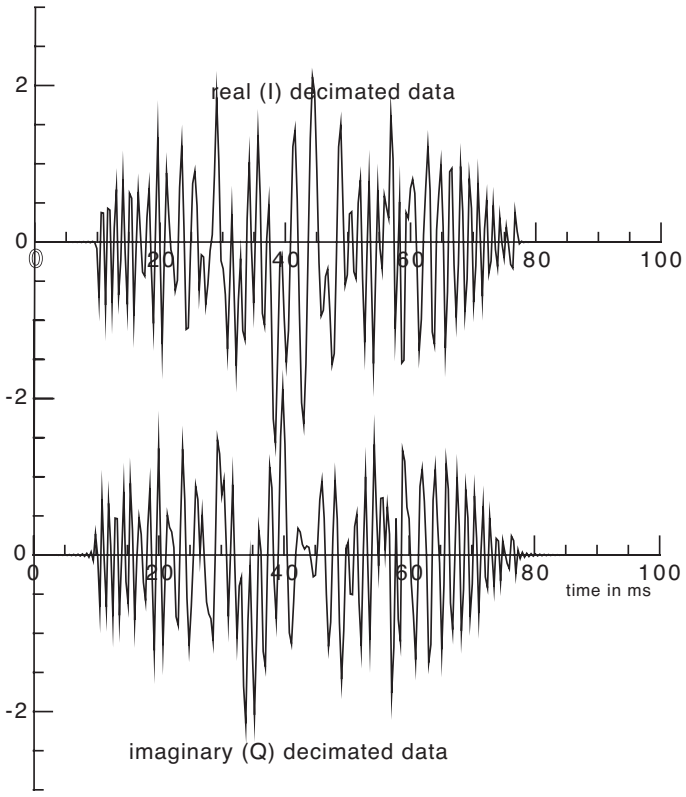

Fig. 6 — The real (*I* or "in phase") and the imaginary (*Q* or "quadrature") part of the quadrature decimated, basebanded data from Fig. 5. This data has been decimated by a factor of 16 so that there are 288 samples in each of the two time series rather than the 4,600 samples in the original 100-ms long real time series.

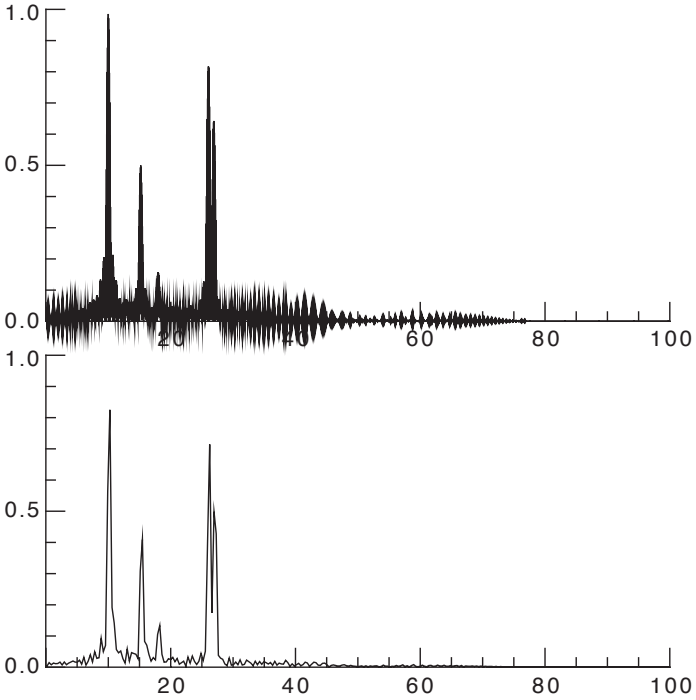

Fig. 7 — The deconvolved data at the original frequency (top) and decimated and basebanded (bottom). The main lobes are 1-ms wide and there are strong side lobes. A time window on the chirp can reduce the side lobes while giving wider main lobes.

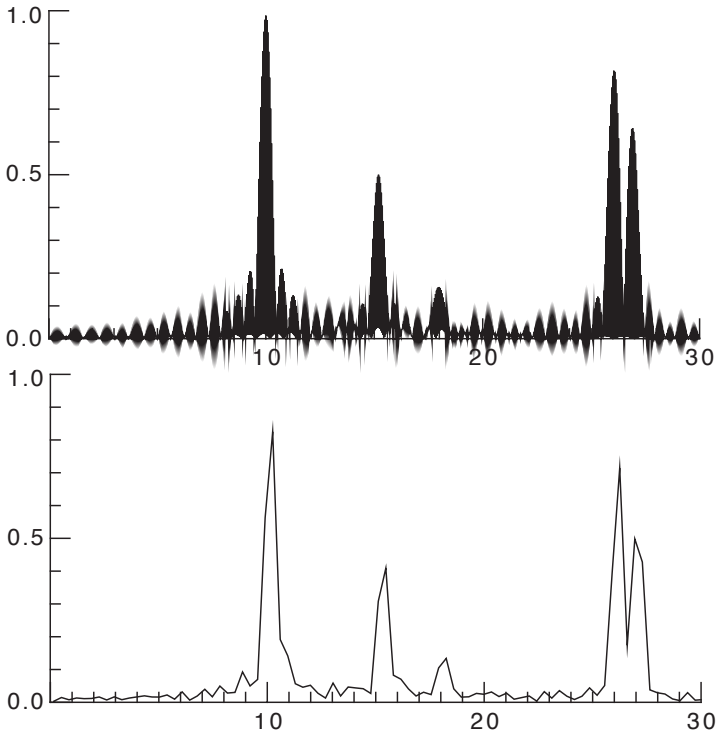

Fig. 8 — Enlargement of the deconvolved original waveform and the decimated, basebanded data. The times and amplitudes agree with the original input parameters.

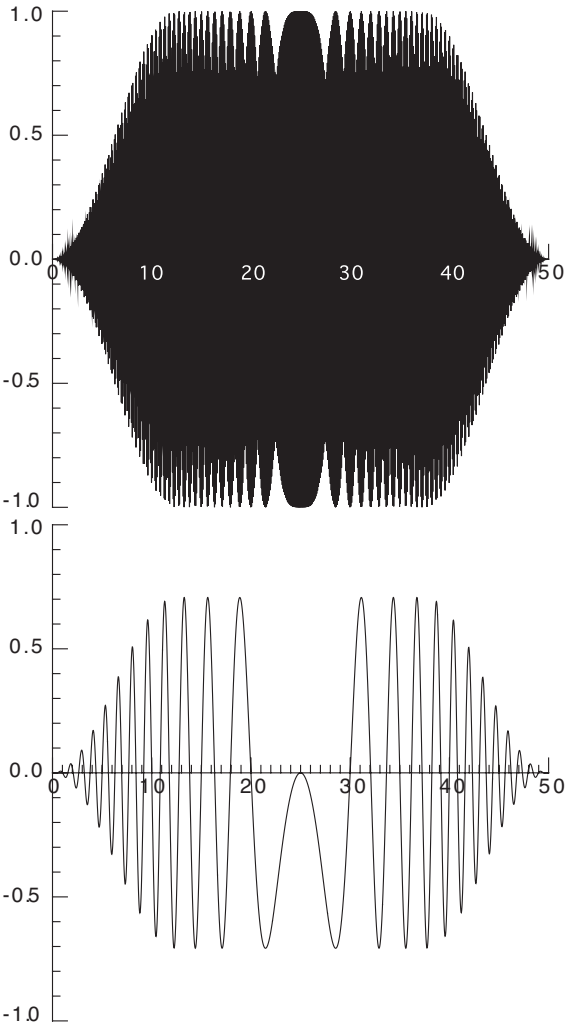

Fig. 9 — The chirp source windowed by a quarter cosine; otherwise the parameters are the same as in Fig. 3

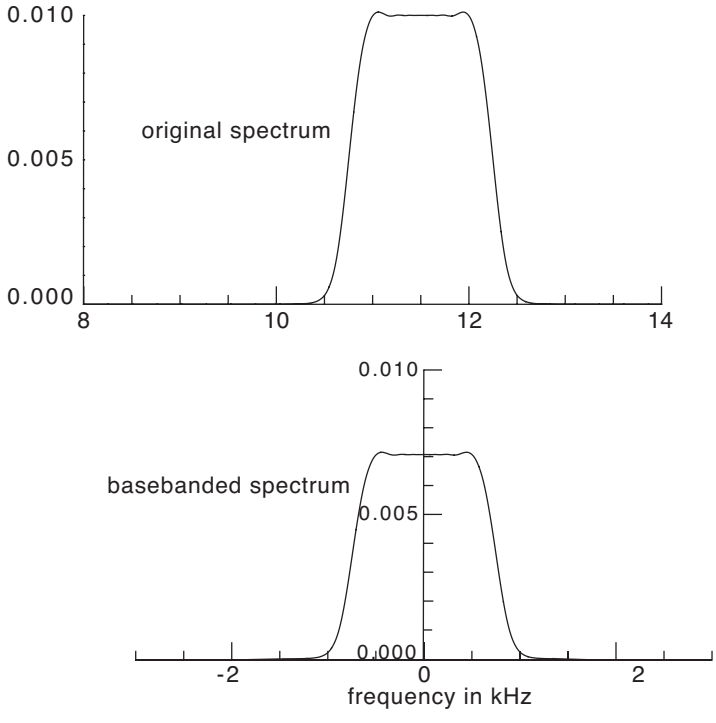

Fig. 10 — Spectra of the windowed linear chirp in the original form (top) and after the basebanding frequency shift (bottom)

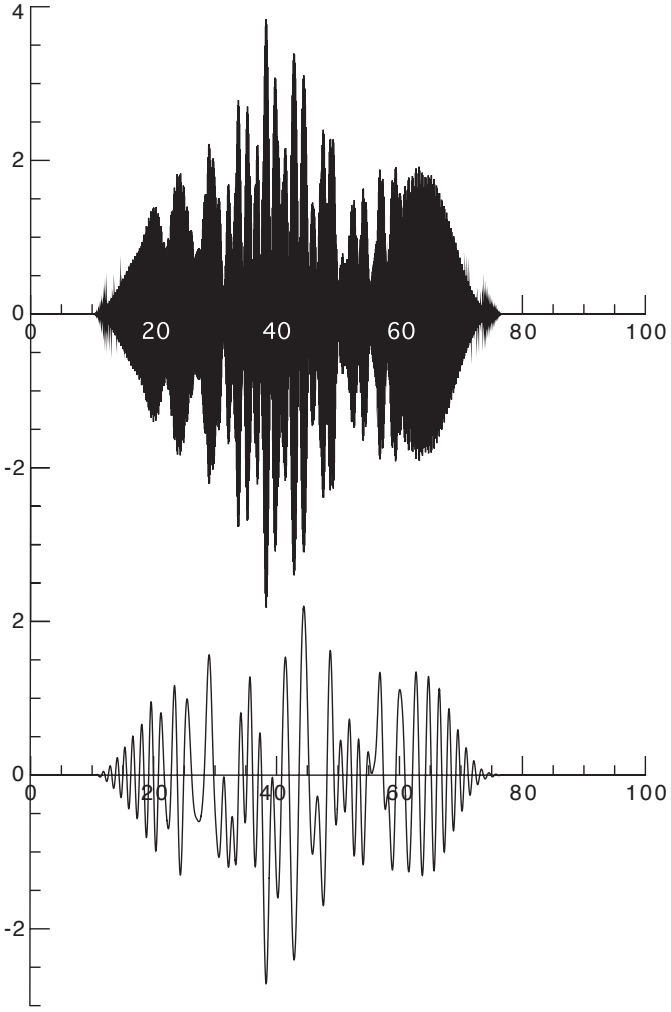

Fig. 11 — The 10.5- to 12.5-kHz data using the windowed chirp from Fig. 9 (top) as in Fig. 5. The response impulses are at 10.0, 15.2, 18.0, 26.0, and 27.0 ms. The basebanded data are shown in the bottom figure.

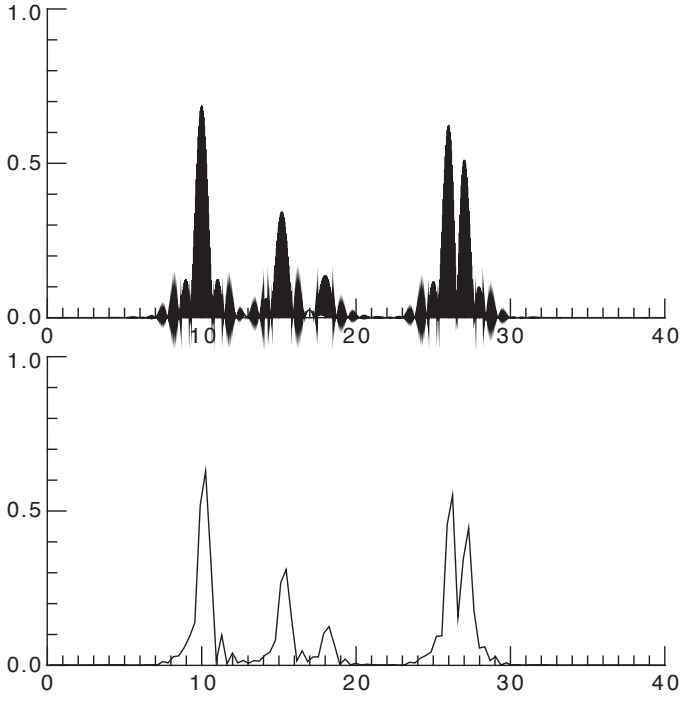

Fig. 12 — The deconvolved response from the original data (top) and from the decimated, basebanded data. The times and relative amplitudes agree with the original input parameters.

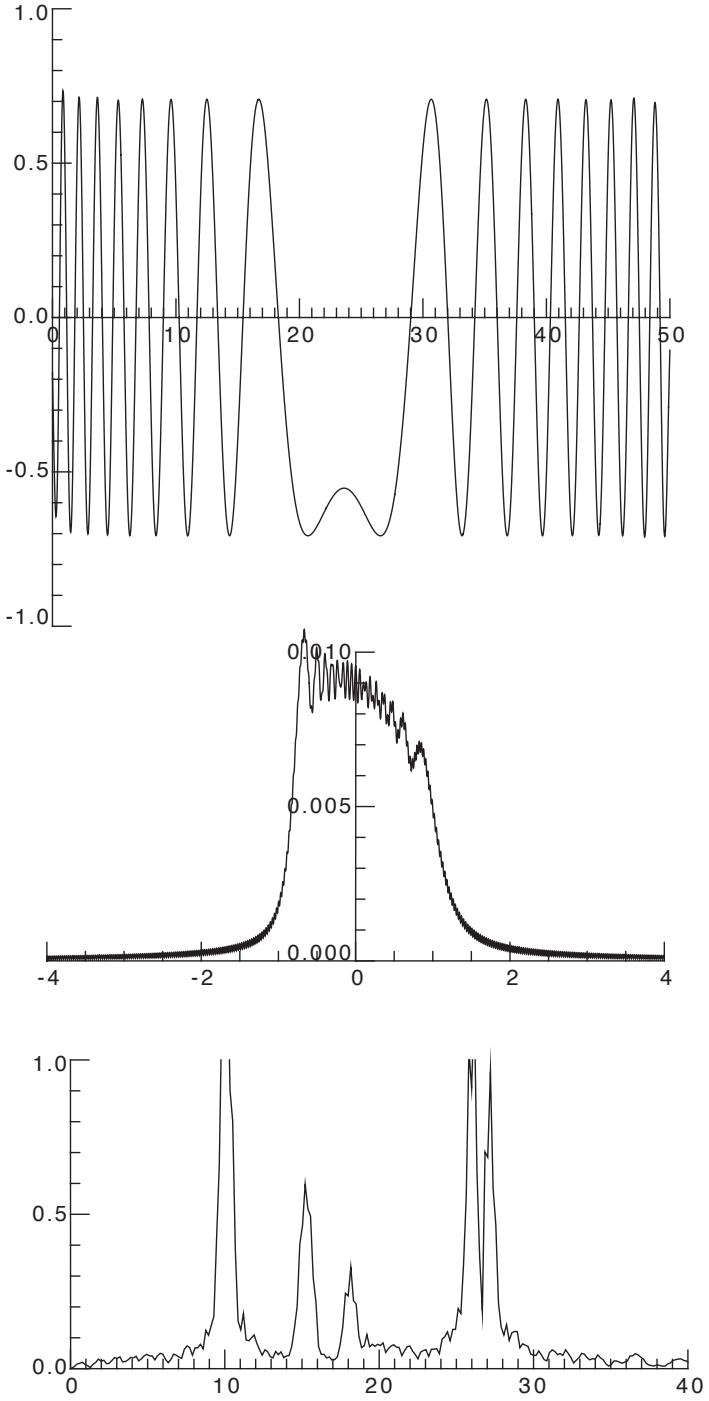

Fig. 13 — The basebanded time series source (top), the basebanded spectrum (middle), and the deconvolved Green's function using this source (bottom)

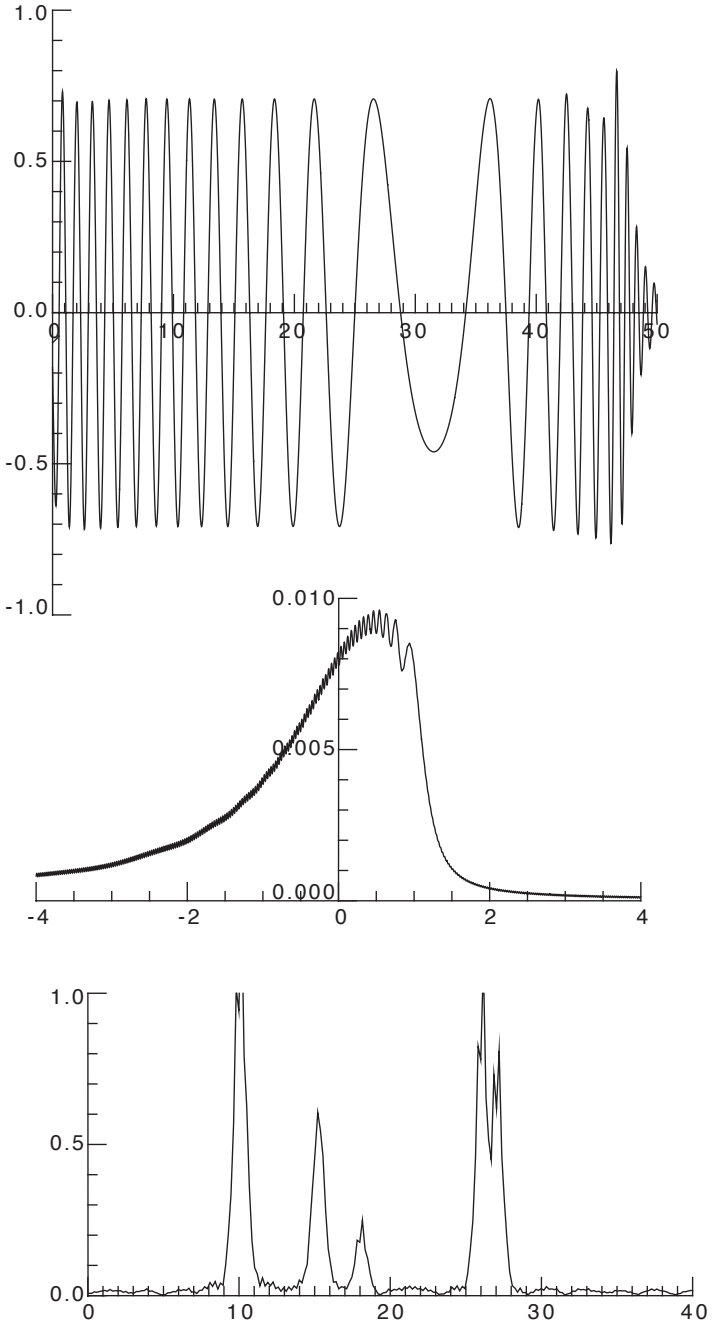

Fig. 14 — The basebanded time series of an alternative, nonlinear chirp signal (top), its spectrum (middle), and the deconvolved Green's function from synthetic data generated with this source

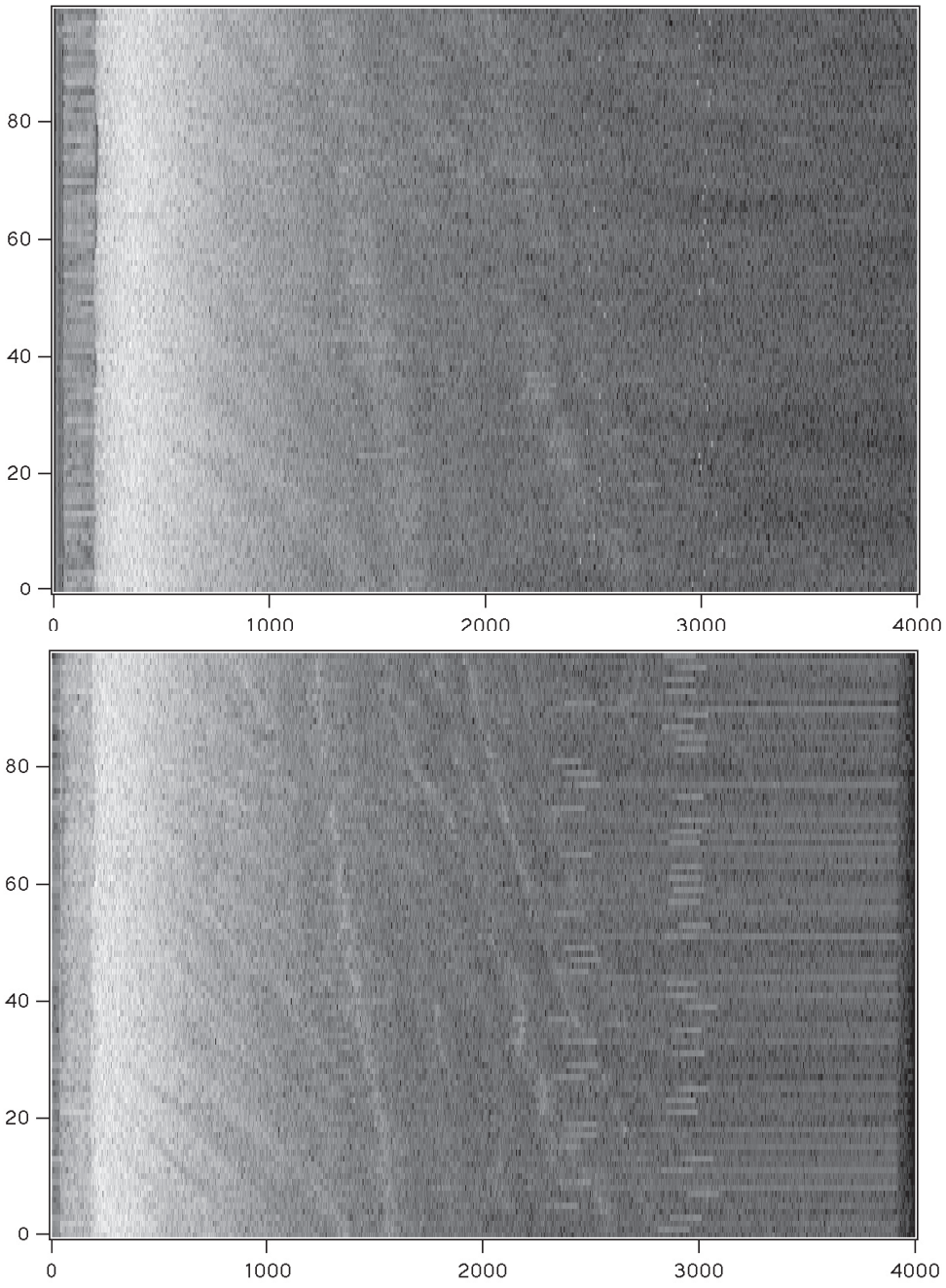

Fig. 15 — Field data from the Norwegian Sea using a chirp signal (top) and after deconvolving the chirp signal (bottom). The horizontal scale is samples and the vertical is traces. The seafloor reflection is near the 200th sample and no time- or angle-dependent gain has been applied.

A nonlinear chirp has a nonuniform frequency change. By selecting certain frequency shift schemes, one can reduce or eliminate the need for a time-domain window. Without a time-domain window, the signal is at full amplitude for its duration and is easier to generate electronically. Two different nonlinear chirps are shown in Figs. 13 and 14, along with their spectra and resulting Green's functions. Both of these are imperfect, as can be seen by their asymmetrical spectra, but the resulting deconvolved Green's functions are superior to the unwindowed, linear chirp solution (Fig. 8). Further development of nonlinear chirps should improve the spectral shapes and reduce the autocorrelation side lobes so that the deconvolved Green's functions will be even better.

The Seamap group from the Naval Oceanographic Office (NAVOCEANO) collected several minutes of chirp data near the Norwegian coast during the summer of 1999. These data were basebanded and quadrature decimated as described in the Synthetic Data section. The transmitted chirp signal was recorded as voltage and current sent to the transducers. The voltage and current signals are somewhat different; the voltage was arbitrarily chosen for use as the transmitted signal. This demonstration only shows that this approach works and is not an optimal implimentation of the chirp nor the deconvolution algorithm.

ACKNOWLEDGMENTS

Thanks to CAPT Charles Hopkins, Program Manager at the Space and Naval Warfare Systems Command (SPAWAR) for his generous support. Ken Sharp, Maurice Thiele, Paula Costello, and G.W. Landrum at NAVOCEANO provided all of the essential technical information concerning the Seamap-C system and collected the field data. Maria Kalcic of the Naval Research Laboratory directed the project, and along with Edit Bourgeois of the University of New Orleans, and Andrew Martinez of Tulane University, New Orleans, Louisiana, provided many helpful ideas concerning the data processing and deconvolution. The translations of the IDL language programs into the C language were done at Planning Systems Inc., Slidell, Louisiana, by Mike Duncan and Pat McDowell. Funding was provided by the SPAWAR program element PMW 185.

REFERENCES

Parent, M.D., C. Fang, T.F. O'Brien, and W.W. Danforth, "First Results of a Deep Tow Chirp Sonar Seafloor Imaging System," Proceedings of the Offshore Technology Conference, OTC 7115, 1993.

Wood, W.T., "Simultaneous Deconvolution and Wavelet Inversion as a Global Optimization," Geophysics, V. 64, p 1108, 1999.

Appendix A

ALGORITHMS IN THE IDL LANGUAGE

This is the file inputa.dat that is read by the synthetic data programs and sets several system and environment parameters.

```
11.5                          ; the central frequency of the chirp in kHz
1.0                           ; half the chirp bandwidth in kHz
50.0                          ; the chirp length in milliseconds
5                             ; the number of reflections (next 2 lines)
10.0,15.2,18.0,26.0,27.0      ; time delay in ms for the reflections
1.0,0.5,0.2,0.8,0.6           ; absolute amplitude of the reflections
16                            ; decimation factor
```

This is the program that generates the synthetic data using the chirp specified in the commented lines.

```
PRO chirp5a
; calculates and plots a chirp signal
; using the system parameters of Seamap C
; (sample rate of 46 kHz)
; includes:       Quadrature modulation
;                 low pass filter
;                 decimated signal
;                 base banding using the center frequency (cfreq)
; reads input parameters from the  file 'input.dat'
; the "data" can have several pulses at delay times
; and amplitudes specified in the file 'input.dat'
; time is in microseconds
; cfreq is center frequency in kilohertz
; del is half bandwidth of chirp (10.5 to 12.5 kHz
;          chirp has cfreq of 11.5 and del of 1.0)
; tlong is chirp length in milliseconds
; dtime is delay to start of chirp in milliseconds
; per is period in time steps (10 µs)
; nar is the time series length

ci = complex(0.0,1.0)
;pi = 3.141592654
pi = !DPI
sqrt2 = sqrt(2)
srate = 46000                     ; the sampling rate
sratek = srate / 1000.            ; the sampling rate in kilohertz
nar = 46000

flts = findgen(nar)
a = fltarr(nar)    ; chirped signal
as = fltarr(nar)    ; chirped source
b = fltarr(nar)    ; real spectrum of chirp signal
t = fltarr(nar)    ; time in microseconds
anal = complexarr(nar)     ; analytic signal
anals = complexarr(nar)     ; analytic source
analf = complexarr(nar)    ; fft of analytic signal
```

```
base = complexarr(nar)     ; base banded signal
bases = complexarr(nar)     ; base banded source
basef = complexarr(nar)    ; fft of base banded signal
basesf = complexarr(nar)    ; fft of base banded source
rbase = fltarr(nar)        ; real part of base banded signal
rbases = fltarr(nar)       ; real part of base banded source
ibase = fltarr(nar)        ; imaginary part of base banded signal
ibases = fltarr(nar)       ; imaginary part of base banded source

t = flts * 1000. / srate

openr, 1, 'inputa.dat'
readf,1,cfreq
readf,1,del
readf,1,tlong
readf,1,nd
dtime = fltarr(nd)
damp  = fltarr(nd)
readf,1,dtime
readf,1,damp
readf,1,decimate
close,1

print,'from inputa.dat', cfreq,del,tlong,nd,decimate
print, 'dtime input',dtime
print, 'damp input',damp
nard = nar / decimate          ; length of decimated time series
nchirp = tlong * srate / 1000.    ; number of samples in chirp
nchirp2 = tlong * srate / 500.    ; twice nchirp
nchirph = tlong * srate / 2000.   ; half nchirp
print,nchirp,nchirp2,nchirph
rbased = fltarr(nard)          ; real, decimated, base banded signal
rbasesd = fltarr(nard)         ; real, decimated, banded source
ibased = fltarr(nard)          ; imaginary, decimated, base banded signal
ibasesd = fltarr(nard)         ; imaginary, decimated, base banded source
td = fltarr(nard)              ; decimated time series (msec / decimate)
tf = t * 46. / 1000. - 23.             ; frequency scale (normal)
print,'tf info',size(tf),tf(0),tf(45999),t(0),t(45999)
wqc = fltarr(nchirp)           ; quarter cosine window for chirp

tend  = float(nar) / float(srate)
tendm = tend * 1000.
dstart = fix(dtime * sratek)
sfreq = cfreq - del
efreq = cfreq + del
print,'tend stuff',tend,tendm,dstart,sfreq,efreq

w = 1.0                        ; default to boxcar window
for i = 0, (nchirp/4)-1 do begin    ; this loop makes a 1/4 cosine window
  wqc(i) = (1./2.) - (1./2.) * cos(pi * i * 4. / nchirp)
  wqc(i + nchirp/4) = 1.0
  wqc(i + nchirp/2) = 1.0
  wqc(nchirp-1-i) = wqc(i)
endfor

;     uncomment the desired line to implement a window
;     and to chose the type of chirp or ping
for i = 0, (nchirp-1) do begin
  ii = float(i)
;  w = 0.5 + 0.5 * cos (pi * 2 * (ii - nchirph) / nchirp)  ; Hanning window
;  w = sin (pi * ii / nchirp)                               ; sine window
```

```
;  w = wqc(i)                                                  ; quarter cosine
;      next line for cw ping
;  per = w*sin(2*pi * (sfreq) *ii/sratek)
;    next line for linear chirp
;  per = w*sin(2*pi * (sfreq + ii *(efreq - sfreq) / nchirp2) *ii/sratek)
;    next line for crude nonlinear chirp
;  per = sin(2*pi*(sfreq+(2*del*acos(1.-ii/nchirp)/pi) ) *ii/sratek)
;    next line for nonlinear chirp tests, this one SEEMS like it
;    should be better
  per = sin(2*pi*(sfreq+del*acos(1.-ii/nchirph)/pi) *ii/sratek)
;  per = sfreq+(del*acos(1.-ii/nchirph)/pi)
;  per = sfreq + ii *(efreq - sfreq) / nchirp2
  as(i) = per
  a(i+dstart(0)) = damp(0) * per
endfor

for n = 1, nd-1 do begin
  for k = 0, (nchirp-1) do begin
    kk = float(k)
;    w = 0.5 + 0.5 * cos (pi * 2 * (kk - nchirph) / nchirp) ; Hanning window
;    w = sin (pi * kk / nchirp)                              ; sine window
;    w = wqc(k)                                               ; quarter cosine
;    next line for linear chirp
;    per = w*sin(2*pi * (sfreq + kk *(efreq - sfreq) / nchirp2) *kk/sratek)
;    next line for crude nonlinear chirp
;    per = sin(2*pi*(sfreq+(2*del*acos(1.-ii/nchirp)/pi) ) *ii/sratek)
;    a(k+dstart(n)) = a(k+dstart(n)) + damp(n) * per
    a(k+dstart(n)) = a(k+dstart(n)) + damp(n) * as(k)
  endfor
endfor

anal = complex(a,hilbert(a,-1))
anals = complex(as,hilbert(as,-1))
analf = 2 * shift(fft(anals,-1), (1 + nar/2))
;analf = 2 * shift(fft(as,-1), (1 + nar/2))
base = sqrt2*a*cos(cfreq*t*2*pi)+ci*sqrt2*a*sin(cfreq*t*2*pi)
bases = sqrt2*as*cos(cfreq*t*2*pi)+ci*sqrt2*as*sin(cfreq*t*2*pi)
;base = anal * exp ( -ci * sfreq * t * 2*pi)
;bases = anals * exp ( -ci * sfreq * t * 2*pi)
;base = anal * exp ( -ci * cfreq * t * 2*pi)
;bases = anals * exp ( -ci * cfreq * t * 2*pi)
;basef = 2 * sqrt(nchirp) * shift(fft(base,-1), (1 + nar/2))
basef = 2 * shift(fft(base,-1), (1 + nar/2))
basesf = 2 * shift(fft(bases,-1), (1 + nar/2))

filt = digital_filter(0,1./decimate,50,nchirph/2)
help,base,bases
rbase = float(base)
ibase = imaginary(base)
rbases = float(bases)
ibases = imaginary(bases)
rbase = convol(rbase,filt,center=1,edge_wrap=1)
ibase = convol(ibase,filt,center=1,edge_wrap=1)
rbases = convol(rbases,filt,center=1,edge_wrap=1)
ibases = convol(ibases,filt,center=1,edge_wrap=1)
help,filt,rbase,ibase,rbases,ibases
print,t(500:504)
print,a(500:504)
print,rbase(500:504)
print,rbases(200:204)
;print,filt(458:462)
```

```
print,float(basesf(2000:2004))

td = rebin(t,nard)
tfd = rebin(tf,nard)
rbased = rebin(rbase,nard)
ibased = rebin(ibase,nard)
rbasesd = rebin(rbases,nard)
ibasesd = rebin(ibases,nard)
print,per,nchirp,t(nar-1)

save, a, filename='original.data'
save, as, filename='original.source'
;save, rbase, filename='basebanded.data'
save, base, filename='basebandedc.data'
;save, rbases, filename='basebanded.source'
save, bases, filename='basebandedc.source'
save, rbased, filename='basebanded_decir.data'
save, ibased, filename='basebanded_decii.data'
save, rbasesd, filename='basebanded_decir.source'
save, ibasesd, filename='basebanded_decii.source'
save, t, filename='time_signal.data'
save, td, filename='time_signal_dec.data'

!p.background=255
!p.color=0
!p.font=0
!p.charsize=1.0
!p.thick=1.0
!p.ticklen=0.03
;!p.region = [0.0,0.0,6.0,11.0]
!p.position = [0.1,0.1,0.9,0.9]
!x.range = [0.0, 50.0]
!y.range = [-1.0, 1.0]

set_plot,'x'
window,2
plot, t, as, subtitle='chirp signal', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0

!y.range = [-1.0, 1.0]
!x.range = [0.0, 50.0]
window,0
;plot, t, abs(base), subtitle='abs of base banded signal', xsty=5, ysty=5
plot, t, rbase, subtitle='base banded signal', xsty=5, ysty=5
;plot, t, a, subtitle='original data', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0

;!x.range = [0.0, 0.2*tendm]
!x.range = [0.0, 50.0]
!y.range = [-1.0, 1.0]
window,3
;plot, t, abs(bases), subtitle='abs of base banded source', xsty=5, ysty=5
;plot, t, filt, subtitle='filter vector', xsty=5, ysty=5
plot, td, rbased, subtitle='decimated I base banded signal', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0

;!y.range = [-1.0, 1.0]
;!x.range = [0.0, 40.0]
```

```
;window,1
;plot, td, ibased, subtitle='decimated Q base banded signal', xsty=5, ysty=5
;plot, t, rbase, subtitle='base banded data', xsty=5, ysty=5
;plot, t, abs(base), subtitle='abs of base banded data', xsty=5, ysty=5
;axis, 0, 0, xax=0
;axis, 0, 0, yax=0

print,max(tf),max(abs(analf))
;!x.range = [-0.1, 0.1]
!y.range = [-0.0, 0.02]
!x.range = [9.0, 14.0]
window,1
;plot, tf, abs(basesf), subtitle='base banded source spectrum', xsty=5, ysty=5
plot, tf, abs(analf), subtitle='original source spectrum', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0

set_plot,'CGM'
;device, binary=1
device, filename='orig_data.cgm'
xs = !d.x_size
ys = !d.x_size
loadct,0
!x.range = [0.0, 100.0]
!y.range = [-3.0, 3.0]
plot, t, a, subtitle='original data', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

device, filename='chirp_signal.cgm'
!x.range = [0.0, 50.0]
!y.range = [-1.0, 1.0]
plot, t, as, subtitle='chirp signal', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

device, filename='base_banded_source.cgm'
!x.range = [0.0, 50.0]
!y.range = [-1.0, 1.0]
plot, t, rbases, subtitle='base banded source', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

device, filename='base_banded_data.cgm'
!x.range = [0.0, 100.0]
!y.range = [-3.0, 3.0]
plot, t, rbase, subtitle='base banded data', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

device, filename='abs_base_banded_data.cgm'
!x.range = [0.0, 40.0]
!y.range = [-1.0, 1.0]
plot, t, abs(base), subtitle='base banded data', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close
```

```
device, filename='real_deci_data.cgm'
!x.range = [0.0, 100.0]
!y.range = [-3.0, 3.0]
plot, td, rbased, subtitle='real (I) decimated data', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

device, filename='imag_deci_data.cgm'
!x.range = [0.0, 100.0]
!y.range = [-3.0, 3.0]
plot, td, ibased, subtitle='imaginary (Q) decimated data', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

device, filename='base_spec.cgm'
!x.range = [-3.0, 3.0]
!y.range = [-0.0, 0.02]
plot, tf, abs(basesf), subtitle='base banded source spectrum', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

device, filename='orig_spec.cgm'
!x.range = [8.0, 14.0]
!y.range = [-0.0, 0.02]
plot, tf, abs(analf), subtitle='original source spectrum', xsty=5, ysty=5
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

;set_plot,'ps'
;device, filename='chirpsignal.ps'
;plot, t, a, subtitle='chirp', xsty=5, ysty=5, /polar
;axis, 0, 0, xax=0, color=0
;axis, 0, 0, yax=0, color=0
;device, /close

END; chirp5a
```

This is the program that read the synthetic data generated by "chirp5a", deconvolves the chirp signal, and makes numerous plots to show the original data, deconvolved data, and their spectra.

```
PRO decon5a

; deconvolves a signal (from the file 'xxx.source')
; from time series data (from the file 'xxx.data')
; using the system parameters of Seamap C
; (sample rate of 46 kHz)
; includes:         reconstructed signal from decimated, I & Q data
;                   low pass filter
;                   decimated signal
; reads input parameters from the  file 'input.dat'
; the "data" can have several pulses at delay times
; and amplitudes specified in the file 'input.dat'
; time is in microseconds

nar = 46000
```

```
srate = 46000                  ; the sampling rate
sratek = srate / 1000.         ; the sampling rate in kilohertz
sratekd = sratek/16.           ; the decimated sampling rate in kHz
srkdh = sratekd / 2.           ; half sratekd, used for band shifting
newbase = complexarr(nar)      ; reconstructed base banded data
newanal = complexarr(nar)      ; reconstructed analytic data
newdata = fltarr(nar)         ; reconstructed real data
rbase = fltarr(nar)
ibase = fltarr(nar)
rbases = fltarr(nar)
ibases = fltarr(nar)
flts = findgen(nar)
ci = complex(0.0,1.0)
;pi = 3.141592654
pi = !DPI
sqrt2 = sqrt(2)

openr, 1, 'inputa.dat'
readf,1,cfreq
readf,1,del
readf,1,tlong
readf,1,nd
dtime = fltarr(nd)
damp  = fltarr(nd)
readf,1,dtime
readf,1,damp
readf,1,decimate
close,1
print,cfreq,del,tlong,nd
print, dtime
print, damp
sfreq = cfreq - del
efreq = cfreq + del
dnorm = float(2 * nar) / (tlong * 46.)
;dnorm = float(nar) / (tlong * 46.)
print,"decon normalizing factor",dnorm
nard = nar / decimate
nard2 = nard * 2
narp = nar * 2                 ; length of padded ts (for FFT decon)
nardp = narp / decimate        ; length of padded ts (for FFT decon)
nard2p = nardp * 2             ; length of padded ts (for FFT decon)
td2 = fltarr(2*nard)
resultbrd = fltarr(nard)      ;
resultbid = fltarr(nard)      ;
resultbd = complexarr(nard)   ;

restore, filename='original.data'
restore, filename='original.source'
;restore, filename='basebanded.data'
restore, filename='basebandedc.data'
;restore, filename='basebanded.source'
restore, filename='basebandedc.source'
restore, filename='basebanded_decir.data'
restore, filename='basebanded_decii.data'
restore, filename='basebanded_decir.source'
restore, filename='basebanded_decii.source'
restore, filename='time_signal.data'
restore, filename='time_signal_dec.data'

tf = t * 46. / 1000. - 23.             ; frequency scale (normal)
tfd = rebin(tf,nard)                   ; frequency scale (decimated)
```

```
rbase  = float(base)
ibase  = float(imaginary(base))
rbases = float(bases)
ibases = float(imaginary(bases))

!p.background=255
!p.color=0
!p.font=0
!p.charsize=1.0
!p.thick=1.0
!p.ticklen=0.03
;!p.region = [0.0,0.0,6.0,11.0]
!p.position = [0.1,0.1,0.9,0.9]
!x.range = [0.0, 100.0]
!y.range = [-3.0, 3.0]

set_plot,'x'
window,2
plot, t, a, subtitle='original signal', xsty=4, ysty=4
axis, 0, 0, xax=0
axis, 0, 0, yax=0

window,0
!x.range = [0.0, 50.0]
!y.range = [-2.0, 2.0]
plot, t, as, subtitle='original source', xsty=4, ysty=4
axis, 0, 0, xax=0
axis, 0, 0, yax=0

window,3
!x.range = [0.0, 100.0]
plot, td, rbased, subtitle='decimated base banded real signal', xsty=4, ysty=4
;plot, t, abs(base), subtitle='base banded signal', xsty=4, ysty=4
axis, 0, 0, xax=0
axis, 0, 0, yax=0

;window,6
;!x.range = [0.0, 100.0]
;plot, td, ibased, subtitle='decimated base banded im signal', xsty=4, ysty=4
;;plot, t, abs(base), subtitle='base banded signal', xsty=4, ysty=4
;axis, 0, 0, xax=0
;axis, 0, 0, yax=0

window,1
!x.range = [0.0, 50.0]
plot, td, rbasesd, subtitle='decimated base banded source', xsty=4, ysty=4
;plot, t, abs(bases), subtitle='base banded source', xsty=4, ysty=4
axis, 0, 0, xax=0
axis, 0, 0, yax=0

;window,0
;!x.range = [0.0, 50.0]
;plot, td, ibasesd, subtitle='decimated base banded im source', xsty=4, ysty=4
;;plot, t, abs(bases), subtitle='base banded source', xsty=4, ysty=4
;axis, 0, 0, xax=0
;axis, 0, 0, yax=0

info = size(a)
print,info
lag = lindgen(info(1)-2)
```

```
;;resultc = fft(fft(a,-1)/fft(as,-1),1)/nar
;resultc = dnorm * c_correlateb(as,a,lag, /covariance)
;
;!x.range = [0.0, 40.0]
;!y.range = [-1.0, 1.0]
;window,6
;plot, t, abs(resultc), subtitle='deconvolved original signal', xsty=4, ysty=4
;;plot, t, resultc, subtitle='deconvolved original signal', xsty=4, ysty=4
;axis, 0, 0, xax=0
;axis, 0, 0, yax=0
;set_plot,'CGM'
;device, filename='decon_orig_signal.cgm'
;xs = !d.x_size
;ys = !d.x_size
;loadct,0
;plot, t, abs(resultc), subtitle='deconvolved original signal', xsty=4, ysty=4
;axis, 0, 0, xax=0
;axis, 0, 0, yax=0
;device, /close

infod = size(rbased)
print,infod
lagd = lindgen(infod(1))
td2 = rebin(t,2*nard)

based = complex(rbased,ibased)
basesd = complex(rbasesd,ibasesd)
;basesd = complex(rbasesd,ibasesd) + 0.00001 * abs(randomn(seed,nard))

basedp = fltarr(nardp)     ; padded chirped signal
basesdp = fltarr(nardp)    ; padded chirped source
basedp(0:nard-1) = based
basesdp(0:nard-1) = basesd
help,basedp,basesdp
print,'del',del,'srkdh',srkdh
;resultbd = fft(fft(based,-1)/fft(basesd,-1),1)/nard
resultbrd = c_correlate(rbasesd,rbased,lagd, /covariance) + $
            c_correlate(ibasesd,ibased,lagd, /covariance)
resultbid = c_correlate(ibasesd,rbased,lagd, /covariance) - $
            c_correlate(rbasesd,ibased,lagd, /covariance)
resultbd = dnorm * complex(resultbrd,resultbid)
resultbrd2 = 2 * rebin(resultbrd,2*nard)
resultbid2 = 2 * rebin(resultbid,2*nard)
;resultbd2 = dnorm * complex(resultbrd2,resultbid2)
;resultbd2 = dnorm * complex(resultbrd2*cos(del*td2*2*pi),resultbid2*sin(del*td2*2*pi)
)
resultbd2 = dnorm * complex(resultbrd2*cos(srkdh*td2*2*pi),resultbid2*sin(srkdh*td2*2*
pi))
;resultbr = c_correlateb(abs(bases),abs(base),lag, /covariance)

set_plot,'x'
!x.range = [0.0, 40.0]
!y.range = [-1.0, 1.0]
window,4
;plot, td, dnorm*resultbrd, subtitle='deconvolved, decimated, base banded', xsty=4,
ysty=4
;plot, td, abs(resultbd), subtitle='deconvolved, decimated, base banded', xsty=4,
ysty=4
plot, td2, abs(resultbd2), subtitle='deconvolved, decimated, base banded', xsty=4,
ysty=4
;plot, t, dnorm * resultbr, subtitle='deconvolved base banded signal', xsty=4, ysty=4
```

```
axis, 0, 0, xax=0
axis, 0, 0, yax=0
set_plot,'CGM'
device, filename='decon_bb_deci.cgm'
;plot, td, abs(resultbd), subtitle='deconvolved, decimated, base banded', xsty=4,
ysty=4
plot, td2, abs(resultbd2), subtitle='deconvolved, decimated, base banded', xsty=4,
ysty=4
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

;   to reconstruct data to original frequency
;rbasen = rebin(rbased,nar)
;ibasen = rebin(ibased,nar)
;rbasesn = rebin(rbasesd,nar)
;ibasesn = rebin(ibasesd,nar)
;   to reconstruct data to twice decimated frequency
rbasen = rebin(rbased,2*nard)
ibasen = rebin(ibased,2*nard)
rbasesn = rebin(rbasesd,2*nard)
ibasesn = rebin(ibasesd,2*nard)

;    reconstruct to original frequency
;newbase = sqrt2*rbasen*cos(cfreq*t*2*pi)+sqrt2*ibasen*sin(cfreq*t*2*pi)
;newbases = sqrt2*rbasesn*cos(cfreq*t*2*pi)+sqrt2*ibasesn*sin(cfreq*t*2*pi)
;    reconstruct to bandwidth frequency
;newbase = sqrt2*rbasen*cos(del*td2*2*pi)+sqrt2*ibasen*sin(del*td2*2*pi)
;newbases = sqrt2*rbasesn*cos(del*td2*2*pi)+sqrt2*ibasesn*sin(del*td2*2*pi)
newbase = sqrt2*rbasen*cos(srkdh*td2*2*pi)+sqrt2*ibasen*sin(srkdh*td2*2*pi)
newbases = sqrt2*rbasesn*cos(srkdh*td2*2*pi)+sqrt2*ibasesn*sin(srkdh*td2*2*pi)
newdata = float(newbase)

set_plot,'x'
!x.range = [0.0, 100.0]
!y.range = [-3.0, 3.0]
window,7
;plot, t, newdata, subtitle='reconstructed data', xsty=4, ysty=4
plot, td2, newdata, subtitle='reconstructed data', xsty=4, ysty=4
axis, 0, 0, xax=0
axis, 0, 0, yax=0
set_plot,'CGM'
device, filename='recon_data.cgm'
plot, td2, newdata, subtitle='reconstructed data', xsty=4, ysty=4
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

info = size(newdata)
print,info
lagn = lindgen(info(1)-2)
resultc = 2 * dnorm * c_correlateb(newbases,newbase,lagn, /covariance)
newbasep = fltarr(nard2p)     ; padded chirped signal
newbasesp = fltarr(nard2p)    ; padded chirped source
newbasep(0:nard2-1) = newbase
newbasesp(0:nard2-1) = newbases + 0.001 * abs(randomn(seed,nard2))
help,newbasep,newbasesp
;resultc = fft(fft(newbase,-1)/fft(newbases,-1),1)/nard2
help,resultc

set_plot,'x'
```

```
!x.range = [0.0, 40.0]
!y.range = [-1.0, 1.0]
window,5
plot, td2, abs(resultc), subtitle='deconvolved reconstructed signal', xsty=4, ysty=4
;plot, t, resultc, subtitle='deconvolved reconstructed signal', xsty=4, ysty=4
;plot, td2, resultc, subtitle='deconvolved reconstructed signal', xsty=4, ysty=4
axis, 0, 0, xax=0
axis, 0, 0, yax=0
set_plot,'CGM'
device, filename='decon_recon_data.cgm'
plot, td2, abs(resultc), subtitle='deconvolved reconstructed data', xsty=4, ysty=4
axis, 0, 0, xax=0
axis, 0, 0, yax=0
device, /close

END; decon5a
```

This program reads the field data and the recorded chirp signal, deconvolves the chirp from the data and writes files that can be plotted as image files.

```
PRO rwseamap

pi = !DPI
short = 4000
srate = 46000                  ; the sampling rate
sratek = srate / 1000.         ; the sampling rate in kilohertz
sratekd = sratek/16.           ; the decimated sampling rate in kHz
srkdh = sratekd / 2.           ; half sratekd, used for band shifting

nsam = long(1)
ping = fltarr(8)

DVbot = fltarr(28000)
DVtop = fltarr(28000)
;DVdata = complexarr(14000)
ints = findgen(14000)
out = fltarr(14000)            ; abs of complex data
;rbs = fltarr(14000)          ; real (I) of source
;ibs = fltarr(14000)          ; imaginary (Q) of source
;rbd = fltarr(14000)            ; real (I) of data
;ibd = fltarr(14000)            ; imaginary (Q) of data
;decon = complexarr(14000)
;deconr = fltarr(14000)
;deconi = fltarr(14000)
;lag = lindgen(14000)
outs = fltarr(short)             ; abs of complex data
rbs = fltarr(short)            ; real (I) of source
ibs = fltarr(short)            ; imaginary (Q) of source
rbd = fltarr(short)              ; real (I) of data
ibd = fltarr(short)              ; imaginary (Q) of data
decon = complexarr(short)
deconr = fltarr(short)
deconi = fltarr(short)
lag = lindgen(short)
td2 = fltarr(2*short)
flts = findgen(short)
t = flts * 1000. / srate
td2 = rebin(t,2*short)
```

```
DVdata = complexarr(short)

dnorm = 0.001

openr, 1, 'port_xmt.bin'
openr, 2, 'port_rcv.bin'
openw, 3, 'outdata.bin'
openw, 4, 'outdecon.bin'

for k = 0, 99 do begin

  readu,1,nsam
  nsamx = 2 * nsam
;  print, nsam, nsamx

  IQVbot = fltarr(nsamx)
  IQAbot = fltarr(nsamx)
  IQVtop = fltarr(nsamx)
  IQAtop = fltarr(nsamx)
  IQsource = complexarr(nsam)

  readu,1,IQVbot
  readu,1,IQAbot
  readu,1,IQVtop
  readu,1,IQAtop
  for i = 0, nsam-1 do begin
    IQsource(i) = complex(IQVbot(2*i),IQVbot(2*i+1))
;    IQsource(i) = complex(IQAtop(2*i),IQAtop(2*i+1))
    rbs(i) = IQVbot(2*i)
    ibs(i) = IQVbot(2*i+1)
  endfor

  readu,2,DVbot
  readu,2,DVtop
;  for i = 0, 13999 do begin
;    DVdata(i) = complex(DVbot(2*i),DVbot(2*i+1))
;  endfor
  for i = 0, short-1 do begin
    DVdata(i) = complex(DVbot(2*i),DVbot(2*i+1))
  endfor
  for i = 0, short-1 do begin
    rbd(i) = DVbot(2*i)
    ibd(i) = DVbot(2*i+1)
  endfor

;  help,rbs
;  help,rbd
;  help,ibs
;  help,ibd
  deconr = c_correlate(rbs,rbd,lag, /covariance) + $
           c_correlate(ibs,ibd,lag, /covariance)
  deconi = c_correlate(ibs,rbd,lag, /covariance) - $
           c_correlate(rbs,ibd,lag, /covariance)
  decon  = dnorm * complex(deconr,deconi)
  resultbrd2 = 2 * rebin(deconr,2*short)
  resultbid2 = 2 * rebin(deconi,2*short)
  resultbd2 = dnorm * complex(resultbrd2*cos(srkdh*td2*2*pi),resultbid2*sin(srkdh*td2*
2*pi))

  print,k,nsam,nsamx,max(abs(IQsource)),max(abs(DVdata)),max(abs(decon))
```

```
  out = 20. * alog10(abs(DVdata) + 0.001)
  writeu,3,out

;  outs = 20. * alog10(abs(decon) + 0.001)
  outs = 20. * alog10(abs(resultbd2) + 0.001)
  writeu,4,outs

endfor

close,1
close,2
close,3
close,4

END
```

Appendix B

ALGORITHMS IN THE C LANGUAGE

This is the file makeData.c

```
/* file: makeData.c
written by: PSI
*/
//TOP
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <malloc.h>
#include “complex.h”
#include “windows.h”
#include “mdsp.h”
#include “fft.h”
#include “zmath.h”
#include “round.h”

void main(int argc, char *argv[]) {


       float pi;
       float sqrt2;
       float sratek;
       float Fs;
       float *b;
       float *t;
       float dt;
       float *dtime;
       float *damp;
       float *fts;
       float *td;
       float *tf;
       float *filt;
       float ftemp;

       float cfreq;
       float del;
       float tlong;
       float sfreq;
       float efreq;
       float w;
       float ii;
       float per;

       float kk;
       float f1;
       float f2;
```

```
//VARS
       int nar;

       long int srate;
       long int nd;
       long int nard;
       long int nchirp;
       long int nchirp2;
       long int j, i, n, k;

       int rsize;
       int itemp;
       int filterSize;
       int decimate;
       int NFFT;

       int *dstart;

       /* Temp arrays for FFTs */
       COMPLEX *FFT0;
       COMPLEX *FFT1;
       COMPLEX *FFT2;

       COMPLEX *CTEMP0;
       COMPLEX *CTEMP1;
       COMPLEX *CTEMP2;

       COMPLEX *a;
       COMPLEX *as;
       COMPLEX *a_c;
       COMPLEX *base;
       COMPLEX *base_c;
       COMPLEX *bases;
       COMPLEX *based;
       COMPLEX *based_c;
       COMPLEX *basesd;


       COMPLEX *cfilt;
       COMPLEX *fftOfFilter;
       COMPLEX *fftOfBBSignal;
       COMPLEX *fftOfBBSource;
       COMPLEX ctemp;
       COMPLEX ctemp2;

       FILE *in;


       /* Initialize variables. */
       pi = 3.14159265358979238462633833f;
       sqrt2 = (float)sqrt(2.0f);
       srate = 46000;
       Fs = srate;
       dt = 1.0/Fs;
       nar = 11500;

       /* Allocate arrays. */

       /* Intitial Source and Signal */
       a = (COMPLEX *)malloc(sizeof(COMPLEX)*nar);
       as = (COMPLEX *)malloc(sizeof(COMPLEX)*nar);
```

```
	/* Float arrays. */
	t = (float *)malloc(sizeof(float) * nar);	/* Time in microseconds. */
	td = (float *)malloc(sizeof(float) * nard);		/* Decimated time seres
msec/decimate. */
	tf = (float *)malloc(sizeof(float) * nar);		/* Frequency scale (normal).
*/

	/* Zero out all arrays. */
	for(j=0;j<nar;j++) {
		a[j] = cmplx(0.0,0.0);
		as[j] = cmplx(0.0,0.0);
		t[j] =  (float) j * dt;
		tf[j] = t[j] * Fs - Fs/2.0;
	}

	/* Read input deck. */
	if (NULL == (in = (fopen("inputa.dat", "r")))) {
		fprintf(stderr,"Error: Could not open file <%s> \n", "inputa.dat");
		exit(0);
	 }

	/* Read values from file. */
	fscanf(in, "%f", &cfreq);  /* Center frequency. */
	fscanf(in, "%f", &del);		/* Bandwidth */
	fscanf(in, "%f", &tlong);  /* Chirp length. */
	fscanf(in, "%d", &nd);		/* # of reflectors */

	/* change units on input values MED */
	cfreq *= 1000.0;	/* kHz to Hz */
	del *= 1000.0;		/* kHz to Hz */
	tlong /= 1000.0;	/* ms to s */
	sfreq = cfreq - del/2;
	efreq = cfreq + del/2;

	/* Allocate dtime, damp and dstart arrays. */
	dtime = (float *)malloc(sizeof(float) * nd);
	damp = (float *)malloc(sizeof(float) * nd);
	dstart = (int *)malloc(sizeof(int) * nd);

	/* Read dtime array. */
	for(j=0;j<nd;j++) {
		fscanf(in, "%f", &dtime[j]);
		dtime[j] /= 1000.0;
		dstart[j] = (int)(dtime[j] * Fs);
	}

	/* Read damp array. */
	for(j=0;j<nd;j++) fscanf(in, "%f", &damp[j]);

	/* Read decimate value and close file. */
	fscanf(in, "%d", &decimate);
	fclose(in);

	/* Define array parameters. */
	nard = nar/(int)decimate;	/* Length of decimated time series. */

	nchirp = (int) (tlong*Fs);	/* no. of samples in chirp */
	nchirp2 = 2*nchirp;

	/* Build the Source Chirp */
	w = 1.0f;
```

```
	for(i = 0; i < nchirp; i++) {
		ii = (float) i;
		per = w*(float)sin( 2.0f*pi*( sfreq + ii * (efreq - sfreq)/nchirp2 ) * ii/
Fs);
		as[i] = cmplx(per,0.0);
	}
	//for(j=0;j<nar;j++) printf("%f\t%f\n", as[j].real, as[j].imag);

	/* Build the return by summing delayed copies of the source */
	for(n = 0; n < nd; n++) {
		for(k = 0; k < nchirp; k++) {
			kk = (float)k;
			per = damp[n]*w*(float)sin(2.0f*pi * (sfreq + kk * (efreq - sfreq)/
nchirp2) * kk/Fs);
			i = k+dstart[n];
			a[i] = cadd( a[i], cmplx(per,0.0) );
		}
	}
	//for(j=0;j<nar;j++) printf("%f\t%f\n", a[j].real, a[j].imag);


	CTEMP0 = (COMPLEX *)malloc(sizeof(COMPLEX)*nar);
	CTEMP1 = (COMPLEX *)malloc(sizeof(COMPLEX)*nar);
	/*  Base Band */
	for(j=0;j<nar;j++){
		ctemp = cexp( cmplx(0.0, 2.0*pi*cfreq*t[j]) );
		ctemp = cmul(ctemp, cmplx(sqrt2,0.0));
		CTEMP1[j] = cmul( a[j], ctemp );
		CTEMP0[j] = cmul( as[j], ctemp );
	}
	//for(j=0;j<nar;j++) printf("%f\t%f\n", bases[j].real, bases[j].imag);
	//for(j=0;j<nar;j++) printf("%f\t%f\n", base[j].real, base[j].imag);

	/* build a 129pt Low Pas FIR filter */
	filterSize = 129;
	filt = fir_low(filterSize, 2.0*del/Fs, HAMMING);
	/* Make the filter complex */
	cfilt = malloc(sizeof(COMPLEX)*filterSize);
	for(j=0;j<filterSize;j++){
		cfilt[j] = cmplx(filt[j],0.0);
	 }
	free(filt);  /* cfilt is the only one we need */

	/* Take everything into the frequency domain */
	NFFT = (int) near2(nar);   /* find nearest power of 2 */


	bases = cconv1d(cfilt,CTEMP0, filterSize, nar);
	base = cconv1d(cfilt, CTEMP1, filterSize, nar);
	rsize = nar+filterSize-1;
	free(CTEMP0);
	free(CTEMP1);

	 basesd = (COMPLEX *)malloc(sizeof(COMPLEX) * nard);
	 based = (COMPLEX *)malloc(sizeof(COMPLEX) * nard);
	itemp = 0;
	ftemp = 1.0/((float)decimate);
	for(j=0;j<nard;j++) {
	   ctemp = cmplx(0.0,0.0);
	   ctemp2 = cmplx(0.0,0.0);
	   for(k = 0 ;k < decimate ;k++) {
```

```
            ctemp = cadd(ctemp, bases[j*decimate+k]);
            ctemp2 = cadd(ctemp2, base[j*decimate+k]);
          }
           basesd[j] = cmul( cmplx(ftemp,0.0), ctemp);
          based[j] = cmul( cmplx(ftemp,0.0), ctemp2);
          itemp++;
       }
       //for(j=0;j<nard;j++) printf("%f\t%f\n",basesd[j].real,basesd[j].imag);
       //for(j=0;j<nard;j++) printf("%f\t%f\n",based[j].real,based[j].imag);

       free(bases);
       free(base);

/*************** NOW FOR DECON *****************/


       /* DCON in FREQUENCY DOMAIN */
       NFFT = 4*near2(nard);
       FFT0 = fft(basesd,nard,NFFT,-1.0,&itemp);
       FFT1 = fft(based,nard,NFFT,-1.0,&itemp);
       for(j=0;j<itemp;j++) {
           FFT1[j] = cmul(FFT1[j],conj(FFT0[j]));
        }
       base_c = fft(FFT1,NFFT,NFFT,1.0,&itemp);
       ctemp = cmplx( 1.0/((float)nar), 0.0);                /* Normalize */
       for(j = 0; j < NFFT; j++) {
             base_c[j] = cmul( ctemp, base_c[j]);
        }
       for(j=0;j<itemp;j++) printf("%f\t%f\n",base_c[j].real,base_c[j].imag);

       /* DCON IN TIME DOMAIN */
       rsize = nard+nard-1;
       CTEMP1 = (COMPLEX *) malloc(sizeof(COMPLEX)*nard);
       for(j=0;j<nard;j++) {
            CTEMP1[j] = conj(basesd[nard-j-1]);
        }
       base_c = cconv1d(based,CTEMP1,nard,nard);
//     for(j=0;j<rsize;j++) printf("%f\t%f\n",base_c[j].real,base_c[j].imag);

       exit(0);




}
```

This is the file complex.c

```
#include <math.h>

struct dcomplex {
  double real;
  double imag;
};

struct complex {
  float real;
  float imag;
};
```

```
typedef struct dcomplex DCOMPLEX;
typedef struct complex COMPLEX;

/**** Double Precision Versions ******************/

/* Make Complex */
DCOMPLEX dcmplx(double a, double b){
  DCOMPLEX c;
  c.real=a;
  c.imag=b;
  return c;
}

/* Addition */
DCOMPLEX dcadd(DCOMPLEX a, DCOMPLEX b){
   DCOMPLEX c;
   c.real = a.real + b.real;
   c.imag = a.imag + b.imag;
   return c;
}

/* Subtraction */
DCOMPLEX dcsub(DCOMPLEX a, DCOMPLEX b){
   DCOMPLEX c;
   c.real = a.real - b.real;
   c.imag = a.imag - b.imag;
   return c;
}

/* Multiplication */
DCOMPLEX dcmul(DCOMPLEX a, DCOMPLEX b){
   DCOMPLEX c;
   c.real=a.real*b.real-a.imag*b.imag;
   c.imag=a.imag*b.real+a.real*b.imag;
   return c;
}

/* multiply DCOMPLEX by real */
DCOMPLEX dcmulr(double a, DCOMPLEX b) {
   DCOMPLEX c;
   c.real = a * b.real;
   c.imag = a * b.imag;
   return c;
}

/* divide DCOMPLEX by real */
DCOMPLEX dcdivr(double a, DCOMPLEX b) {
   DCOMPLEX c;
   c.real =  b.real / a;
   c.imag =  c.imag / a;
   return c;
}

/* Conjugation */
DCOMPLEX dconj(DCOMPLEX z){
 DCOMPLEX c;
 c.real = z.real;
 c.imag = -z.imag;
 return c;
}
```

```
/* Exponentiation */
DCOMPLEX dcexp(DCOMPLEX z){
  DCOMPLEX c;
  double x,y,temp1,temp2;
  x = z.real;
  y = z.imag;
  temp1 = exp(x)*cos(y);
  temp2 = exp(x)*sin(y);
  c = dcmplx(temp1,temp2);
  return c;
}


/* Magnitude */
double dcabs(DCOMPLEX z){
   double x,y,ans,temp;

   x=abs(z.real);
   y=abs(z.imag);
   ans = sqrt(x*x+y*y);

//   if(x==0.0) ans = y;
//   else if (y == 0.0) ans = x;
//   else if (x > y) {
//      temp=y/x;
//      ans=x*sqrt(1.0+temp*temp);
//   }
//   else {
//      temp=x/y;
//      ans=y*sqrt(1.0+temp*temp);
//   }
   return ans;
}
/*********************************************/
/*** Single Precision versions ***************/
/*********************************************/

COMPLEX cmplx(float a, float b){
  COMPLEX c;
  c.real=a;
  c.imag=b;
  return c;
}

COMPLEX cadd(COMPLEX a, COMPLEX b){
   COMPLEX c;
   c.real = a.real + b.real;
   c.imag = a.imag + b.imag;
   return c;
}

COMPLEX csub(COMPLEX a, COMPLEX b){
   COMPLEX c;
   c.real = a.real - b.real;
   c.imag = a.imag - b.imag;
   return c;
}

COMPLEX cmul(COMPLEX a, COMPLEX b){
   COMPLEX c;
   c.real=a.real*b.real-a.imag*b.imag;
```

```
   c.imag=a.imag*b.real+a.real*b.imag;
   return c;
}

/* multiply DCOMPLEX by real */
COMPLEX cmulr(float a, COMPLEX b) {
   COMPLEX c;
   c.real = a * b.real;
   c.imag = a * b.imag;
   return c;
}

/* divide DCOMPLEX by real */
COMPLEX cdivr(float a, COMPLEX b) {
   COMPLEX c;
   c.real =  b.real / a;
   c.imag =  c.imag / a;
   return c;
}

COMPLEX conj(COMPLEX z){
 COMPLEX c;
 c.real = z.real;
 c.imag = -z.imag;
 return c;
}

COMPLEX cexp(COMPLEX z){
  COMPLEX c;
  float x,y,temp1,temp2;
  x = z.real;
  y = z.imag;
  temp1 = exp(x)*cos(y);
  temp2 = exp(x)*sin(y);
  c = cmplx(temp1,temp2);
  return c;
}


float cabs(COMPLEX z){
   float x,y,ans,temp;

   x=abs(z.real);
   y=abs(z.imag);

   if(x==0.0) ans = y;
   else if (y == 0.0) ans = x;
   else if (x > y) {
      temp=y/x;
      ans=x*sqrt(1.0+temp*temp);
   }
   else {
      temp=x/y;
      ans=y*sqrt(1.0+temp*temp);
   }
   return ans;
}
```

This is the file complex.h

```
#include <math.h>

struct complex {
  float real;
  float imag;
};

struct dcomplex {
  double real;
  double imag;
};

typedef struct dcomplex DCOMPLEX;
typedef struct complex COMPLEX;

DCOMPLEX dcmplx(double a, double b);
DCOMPLEX dcadd(DCOMPLEX a, DCOMPLEX b);
DCOMPLEX dcsub(DCOMPLEX a, DCOMPLEX b);
DCOMPLEX dcmul(DCOMPLEX a, DCOMPLEX b);
DCOMPLEX dcmulr(double a, DCOMPLEX b);
DCOMPLEX dcdivr(double a, DCOMPLEX b);
DCOMPLEX dconj(DCOMPLEX z);
DCOMPLEX dcexp(DCOMPLEX z);
double dcabs(DCOMPLEX z);


COMPLEX cmplx(float a, float b);
COMPLEX cadd(COMPLEX a, COMPLEX b);
COMPLEX csub(COMPLEX a, COMPLEX b);
COMPLEX cmul(COMPLEX a, COMPLEX b);
DCOMPLEX cmulr(float a, COMPLEX b);
DCOMPLEX cdivr(float a, COMPLEX b);
COMPLEX conj(COMPLEX z);
COMPLEX cexp(COMPLEX z);
float cabs(COMPLEX z);
```

This is the file conv1d.c

```
#include <malloc.h>
#include “complex.h”

float *conv1d(float *x, float *y, int Nx, int Ny) {

       register int i,j;
       float *z;
       int Nz;

       Nz = Nx + Ny - 1;
       z = malloc(sizeof(float)*Nz);

       for(j=0;j<Ny;j++) {
         for(i=0;i<Nx;i++){
```

```
          z[i+j] += (x[i]*y[j]);
        }
      }

      return z;

}


COMPLEX *cconv1d(COMPLEX *x, COMPLEX *y, int Nx, int Ny) {

      register int i,j;
      COMPLEX *z;
      COMPLEX ctemp;
      int Nz;

      Nz = Nx + Ny - 1;
      z = (COMPLEX *)malloc(sizeof(COMPLEX)*Nz);

      for(j=0;j<Ny;j++) {
        for(i=0;i<Nx;i++){
          ctemp = cmul(x[i],y[j]);
          z[i+j] = cadd(z[i+j],ctemp);
        }
      }

      return z;

}
```

This is the file fft.c

```
#include <math.h>          /* pow and others */
#include <malloc.h>
#include <string.h>
#include <memory.h>
#include “zmath.h”
#include “round.h”
#include “complex.h”


/* most compilers have PI defined somewhere in math.h.  With gcc
   it’s defined as M_PI.  Rather than try and figure it out what
   the name actually is we’ll just redefine it if it’s missing */

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define TRUE (1)
#define FALSE (0)

//int near2(int N);              /* next highest power of 2 */
//double dlogN(double x, int N);        /* log base N */
//double nint(double x);                /* nearest integer */
//double ceil(double x);                /* next highest integer */

COMPLEX *fft(COMPLEX *data, int Nx, int NFFT, float iSign, int *N) {
```

```
   register int i,j,k;

   COMPLEX *F;                    /* the FFTd data */
   COMPLEX W;              /* complex factor */
   COMPLEX ctemp;          /* temporary complex value */
   COMPLEX cnorm;

   float norm;

   int m,l,istep;
   int p,q;
   int *swap_tbl;


//   NFFT = near2(NFFT);
//   if(NFFT < Nx) NFFT = near2(Nx);                  /* NFFT wasn't specified or is < Nx
*/

   *N = NFFT;                                    /* Return the FFT size as N */

   F = (COMPLEX *)malloc(NFFT*sizeof(COMPLEX));                    /* the resorted array
*/
   for(j=0;j<NFFT;j++){
        F[j] = cmplx(0.0,0.0);           /* initialze F */
   }
   memcpy(F,data,Nx*sizeof(COMPLEX));

   swap_tbl = malloc(NFFT*sizeof(int));


/* NEW GOOD WAY OF BIT REVERSING */
    p = NFFT/2;
    q = 1;
    swap_tbl[0] = 0;
    while(p >= 1) {

      for(i=0; i < q; ++i)
         swap_tbl[i + q] = swap_tbl[i] + p;

      p /= 2;
      q *= 2;
    }

    for(i=0;i<NFFT;++i) {
       ctemp = F[i];
       q = swap_tbl[i];
       if(i < q) {
         F[i] = F[q];
         F[q] = ctemp;
       }
    }

  /* Ok. Now compute the FFT */

  l = 1;
  while(l < NFFT) {
    istep = 2*l;
    for(m = 0; m < l; m++) {
      W = cexp( cmplx(0.0,iSign*M_PI*m/l) );
      for(i = m; i < NFFT; i+=istep) {
        ctemp = cmul(F[i+l],W);
```

```
        F[i+l] = csub(F[i],ctemp);
        F[i] = cadd(F[i],ctemp);

      }
    }
    l = istep;

  }

  return F;


}
```

This is the file fft.h

```
void fork_calc(int lx, COMPLEX *cx, float signi);
DCOMPLEX *dfft(DCOMPLEX *data, int Nx, int NFFT, float iSign,int *N);
COMPLEX *fft(COMPLEX *data, int Nx, int NFFT, float iSign,int *N);
```

This is the file fir_low.c

```
#include <math.h>
#include <malloc.h>
#include "windows.h"


#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define RECT   (0)
#define HANN  (1)
#define HAMMING      (2)
#define BLACK        (3)

float *fir_low(int N, float w0, int win) {

        register int j;
        float *x;               /* the Filter */
        float *w;               /* window */
        int q;

        if(N % 2) N--;
        q = N / 2;
```

```
        x = malloc(sizeof(float)*(N+1));

        for(j=0;j<(N+1) ;j++) {
       x[j] = w0 * (float)(j - q);
        }

        sinc(x,N+1,1);

        switch(win) {
        case RECT:
                w = rect(N+1);
                break;
        case HANN:
                w = hann(N+1);
                break;
        case HAMMING:
                w = hamming(N+1);
                break;
        case BLACK:
                w = blackmann(N+1);
                break;
        default:
                w = rect(N+1);
                break;
        }

        for(j=0;j<(N+1);j++) {
          x[j] = x[j]*w0*w[j];
        }

        return x;
}
```

This is the file mdsp.h

```
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define RECT  (0)
#define HANN  (1)
#define HAMMING      (2)
#define BLACK (3)


//DCOMPLEX *dfft(DCOMPLEX *data, int Nx, int NFFT, float iSign,int *N);
//COMPLEX *fft(COMPLEX *data, int Nx, int NFFT, float iSign,int *N);


float *conv1d(float *x, float *y, int Nx, int Ny);
```

```
COMPLEX *cconv1d(COMPLEX *x, COMPLEX *y, int Nx, int Ny);
float *fir_low(int N, float w0, int win);
float *sinc(float *x, int Nx, int inPlace);

float *rect(int N);
float *hamming(int N);
float *hann(int N);
float *blackmann(int N);
```

This is the file round.c

```
double nint (double x) {

  double frac;
  double whole;
  double y;

  frac = modf(x,&whole);
  y = (frac < 0.5) ? whole : whole + 1;
  return y;
}

double floor (double x){
    long int n;
    double k;
    n = (long) x;
    if ((x-n) < 0) n--;
    k = (double) n;
    return k;

}

double ceil (double x) {
    long int n;
    double k;
    n = (long) x;
    if((x-n) > 0) n++;
    k = (double) n;
    return k;
}
```

This is the file round.h

```
#include <math.h>

double nint (double x);                 /* round to nearest integer */
double floor (double x);                /* round toward -infinity */
double ceil (double x);                 /* round toward infinity */
```

This is the file sinc.c

```
#include <math.h>
#include <malloc.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif


float *sinc(float *x, int Nx, int inPlace) {

/**** build a SINC function from the array x */
/*     x is an array of floating point numbers */
/*     Nx is the number of elements in x.  */
/*     inPlace is a flag */
/*
/*     Makes sin(x)/x function.  If inPlace > 0,
       The operation is done in place and a pointer
       to x is returned.  Otherwise a pointer to
       a newly allocated array is returned
*/

       int j;
       float *y;

       /* if we're doing it in place make y point at x */
        if(inPlace)  y = x;
       /* otherwise allocate a new array */
       else   y = malloc(sizeof(float)*Nx);

       /* make the sinc function and catch zero divides */
       for(j = 0; j < Nx; j++) {
         if(x[j] == 0) y[j] = 1;
          else y[j] = sin(M_PI*x[j])/(M_PI*x[j]);
        }

       return y;

}
```

This is the file windows.c

```
#include <math.h>
#include <malloc.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif


/* N-point Hamming window */
float *hamming(int N) {
       register int j;
       float *w;
       float Nw;
       Nw = (float) (N-1);
       w = malloc(sizeof(float)*N);
```

```
       for(j=0;j<N;j++) {
         w[j] = 0.54 - 0.46*cos( 2.0*M_PI*((float)j)/Nw );
       }
       return w;
}

/* N-point Hann window */
float *hann(int N) {
       register int j;
       float *w;
       float Nw;
       Nw = (float) (N+1);
       w = malloc(sizeof(float)*N);
       for(j=0;j<N;j++) {
            w[j] = 0.5*(1.0 - cos( 2.0*M_PI*(float)(j+1)/Nw ));
       }
       return w;
}



/* N-point Blackmann-Tukey window */
float *blackmann(int N) {
       register int j;
       float *w;
       float Nw;
       Nw = (float) (N-1);
       w = malloc(sizeof(float)*N);
       for(j=0;j<N;j++) {
          w[j] = 0.42 - 0.5*cos( 2.0*M_PI*(float)j/Nw )  +
                 0.08*cos( 4.0*M_PI*(float)j/Nw );

       }
       return w;
}

/* N-point Rectangle window */
float *rect(int N) {
       register int j;
       float *w;
       w = malloc(sizeof(float)*N);
       for(j=0;j<N;j++)  w[j] = 1.0;
       return w;
}
```

This is the file windows.h

```
float *rect(int N);
float *hamming(int N);
float *hann(int N);
float *blackmann(int N);
```

This is the file zmath.c

```
#include <math.h>
```

```
#include <round.h>


/*************************************************************
  Find the next highest number which is a power of 2.  Used
  here to find an acceptable length for the FFT if the
  length of the input array isn't a power of 2.
*************************************************************/

int near2(int N){
    double n=0;
    int Q;
    while( ((double)N/pow(2.0,n)) > 1.0 ) n++;
    Q = (int) nint(pow(2,n));
    return Q;

}

/*************************************************************
  Log base N of a double.  Watch out for 0.
*************************************************************/

double dlogN(double x, int N){

    double y;
    y = log(x)/log(N);
    return y;

}
```

This is the file zmath.h

```
int near2(int N);
double dlogN(double x, int N);
```